# A Communication Library Using Active Messages to Improve Performance of PVM

**Krishnan R Subramaniam*, Suraj C Kothari* and Don E. Heller†**

**Department of Computer Science, Iowa State University, Ames, Iowa 50011, USA*

*and*

*†Ames Laboratory, Ames, Iowa 50011, USA*

**Abstract**

**We present a communication library to improve performance of PVM. The new library introduces communication primitives based on Active Messages. We propose a hybrid scheme that includes a signal driven message notification scheme plus controlled polling. The new communication library is tested along with the normal PVM library to assess the improvement in performance.**

## 1. INTRODUCTION

This paper describes an enhancement to the PVM communication library to improve performance. The new communication library (PVM-AM) is based on the concept of *Active Messages (AM*) [11]. Recent research has shown that communication using active messages has a lower message passing latency compared to other communication schemes [6, 11]. Implementations of active messages on different platforms including, a set of workstations connected by FDDI [6], CM-5 [5], and more recently on the Meiko-CS2 [3] have shown that AM has the potential to provide an order of magnitude reduction in message passing latency over the existing communication schemes. In [7] Dongarra examines the feasibility of adding an AM-layer to PVM, and concludes that a carefully implemented version of AM will provide an improvement over the existing communication primitives in the PVM. This paper describes an effective implementation of AM for reducing the communication latency in PVM environment.

Our communication library, PVM-AM, is based on the same principle as Active Messages. In an Active Messages based scheme, messages are received immediately upon their arrival and an associated function is invoked, which consumes the message (i.e., performs some computation based on the content of the message). The difference between AM and our approach is that we implement the same principles as AM but on a higher network protocol level, which makes it less dependent on specific hardware and hence more portable. In our communication scheme, messages are received immediately on their arrival at the network interface. The message is then stored in the application's memory with some additional information to allow the application process to use the message at a later point in time.

In contrast with PVM-AM, PVM receives messages only when the user-application executes a receive function call. In PVM, if the sender tries to send a message and the user application on the remote machine is not ready to receive it, then the sender buffers the message and subsequently retransmits it. Later, when the receiver executes a receive function call, the receiver may be blocked as the sender is yet to re-transmit the message. This type of blocking, encountered in PVM, is avoided by our communication scheme because the message is received the first time it is sent. In [10] it is shown that draining the network, i.e., receiving messages as quickly as possible when they arrive, is an important factor in reducing message passing latency. To reduce the number of situations where an application process gets blocked in a receive call, is all the more important in the current version of PVM as it does not support multi-threaded applications. The future versions of PVM are expected to support multi-threaded applications [1]. The PVM-AM communication library will be a complement to the threaded versions of PVM.

In order to receive messages as soon as they arrive at the network interface, we need a notfication mechanism to indicate the arrival of a new message. Typical implementations of AM use either interrupt based or polling based notification mechanisms [4, 3]. Unix signals can be used to develop an interrupt driven message noti-

fication scheme for PVM. To deal with the high cost of signal handling in Unix, our message notification scheme uses a combination of signal handling and polling. The signal handler receives the pending message. Subsequently, a communication thread is invoked to poll for a specified number of times. This scheme described in detail in the paper, is well suited for applications where messages arrive in bursts.

The AM-communication library was tested using communication intensive and computation intensive applications. We measured the total time spent in all the message receive function calls and also the overall execution time for each application. The experiments were conducted on a set of HP9000/800 workstations connected by a 10Mbps Ethernet network.

The remainder of the paper is organized as follows. Section 2 gives an overview of PVM and communication issues. Section 3 describes our scheme for efficient communication in PVM. Section 5 provides performance results, and the Appendix describes the programmer's interface to PVM-AM.

## 2. OVERVIEW OF PVM

The PVM message passing system consists of a daemon process and a set of communication primitives. PVM provides the standard message passing routines like *pvm send*( ), which is a non-blocking send, and *pvm recv*( ) which is a blocking *receive*. PVM also provides primitives to start tasks at a remote node, add/delete hosts to/from the current set of machines etc.[2]. PVM messages are tagged messages, i.e., each message is associated with a "tag" defined by the sender. Tagged messages enable a receiver to receive messages of a particular type. Communication between tasks is established using UDP sockets [8].

The daemon process runs on each PVM host machine. The daemons communicate among themselves to perform operations like starting up a user task, multicasting messages, and finding the status of a particular task on a particular host. Tasks have two modes to establish communication with other tasks, the *task-to-task* mode and *task-to-daemon-to-daemon-to-task* mode. In the task-to-task mode the tasks have a direct link to each other using a separate socket. Communication is usually faster in this mode. In the task-daemon-daemon-task communication mode a message from task $T$ running on host $H$ to task $T'$ on host $H'$ takes the logical route from $T$ —> $H$ ==> $H'$ —> $T'$. This communication mode is usually used if the tasks cannot open a dedicated socket to communicate between themselves (this is the case in some Unix systems which have a limit on the number of sockets that can be active at a time).

All the performance comparisons in our paper used the fastest mode of communication in PVM which is the task-to-task mode (the PvmRouteDirect option) and use PvmDataRaw as the packing option (The PvmDataRaw option reduces the time needed to "pack" data if communication is between similar machines).

### 2.1 Performance problems in PVM

PVM provides a clean abstraction to the programmer but the message passing latency is higher than the physical network's latency. In many cases the PVM communication library achieves only 15%- 20% of the network's theoretical capacity [9]. The extra latency is partially due to the overheads involved in TCP/IP communication [8], e.g. the UDP protocol computes a checksum for every packet sent out. The complex message buffering scheme in PVM also lowers the performance.

Various solutions have been proposed to reduce the discrepancy between the network's physical latency and the actual message passing latency observed while using communication protocols. Some of the solutions proposed are, memory mapping of the network buffers directly into user-space [7], using a DLPI (Data Link Provider Interface), writing a device driver for the network interface which can be controlled by the user [11]. These approaches either involve extensive patches to the existing kernel [11], or require root-level permissions for the user [7], or compromise the security of the network [7]. Another problem which increases the message passing latency is that the sender may have to queue up messages to be sent if the receiver is not ready to receive the messages [11]. A typical sequence of events would be :

- *Sender* packs a message into a memory buffer.
- *Sender* initiates a write to a socket to transmit the message.
- *Sender* is NOT able to transmit all of the message as the remote machine's network buffers are full.
- *Sender* has to queue the unsent fragments of the

message.
• *Sender* has to retry transmitting the message, through the PVM daemon.

The operating system's network buffers are quite small compared to the size of messages exchanged using PVM. This means that almost always a message will have to be sent through the PVM daemon. Later, when the *Receiver* executes a receive message operation the following sequence of events takes place :

• *Receiver* initiates a receive message call.
• *Receiver* receives the initial portion of the message written by the Sender in step 2.
• *Receiver* waits for Sender to retransmit the remaining fragment of the message.

The *Receiver* has to wait until the *Sender* transmits the message. The time the *Receiver* spends waiting for the message is called the blocked time during which the *Receiver* is idle. Section 3 discusses our solution to minimize the blocked time.

## 3. COMMUNICATION SCHEME BASED ON ACTIVE MESSAGES

In a typical scenario, our communication scheme will receive a message immediately on arrival, thus ensuring that the Sender does not have to queue up the message. The sequence of actions that take place when a message is sent using our scheme is as follows:

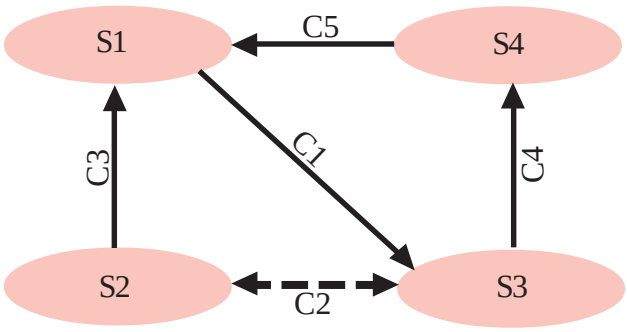
1. Sender executes **pvm_active_ send( )** function.
2. PVM-AM library sends the message out through the network.
3. The message notification mechanism detects the arrival of a message.
4. The message is immediately received into user-memory area.

An efficient message notification (or detection through polling) mechanism is critical to the implementation of Active Messages. One could either poll for a message every so often or setup a signal to notify the arrival of a message. Most Unix systems do not give the user direct control of the machine level interrupts, they merely provide an interface to interrupts through Unix signals. Many Unix systems do not implement a "reliable" version of signals [8], i.e., if two signals arrive in quick succession, one could lose one of the signals. This means that if we were to choose signals as a message notification mechanism we could lose some messages. If we choose to poll for messages, we need to decide how often to poll and we could be wasting time by polling regularly when there are no messages to be received.

Our notification mechanism performs polling for a short period of time after every message is received. If no message is received within this period, we disable polling and enable the interrupt driven message notification mechanism. To be able to efficiently control the communication operations, the user process is split into two threads, a computation thread for performing the useful computation required by the application, and a communication thread to handle communication. The *send* operations are performed in the computation thread itself as we don't need to poll the network interface for the *send* operation. The communication thread polls the network interface waiting for a message to arrive. A timer which generates a SIGALRM signal is used to switch regularly between the communication and computation threads. Regular switching between the two threads ensures that messages are received quickly, but it wastes CPU cycles when there is no message to be received. To reduce these wasted polls, we disable switching between the threads if we have not received a message after $N_{poll}$ switches. We have to restart switching between threads if we detect a message (in order for the message to be received by the communication thread). However, by disabling the regular switching between communication and computation threads we have also disabled our message notification mechanism. The problem now is how do we know when a message has arrived if we disable polling? Our solution is to use a signaling mechanism to notify the arrival of a message on a socket, by setting up the sockets to generate a SIGIO signal [8] if there is a message to be read. The signal handler for SIGIO moves the flow of execution to the communication thread and also restarts the regular switching between the computation and communication threads (i.e. the polling mechanism). The coordination between the the timer and SIGIO signal handlers is best explained using the following logic for the two signal handlers along with the state diagram in Figure 1. The signal handlers work as follows :

*Timer (SIGALRM) signal handler*
– If no message has been received by communication thread, increment *counter*
– If a message was received by communication thread then reset *counter* to 0.

– If counter > $N_{poll}$ then disable timer, and enable SIGIO signal handler




**STATES**
S1 = Compute thread with only SIGIO enabled
S2 = Compute thread with only polling enabled
S3 = Receive a message
S4 = Compute with SIGIO & polling disabled

**CONDITIONS**
C1 :: SIGIO raised by network device
C2 :: Thread switch during polling
C3 :: *counter* > $N_{poll}$ in SIGALRM handler
C4 :: # of Buffered messages > Threshold
C5 :: # of Messages < Threshold

**Figure 1. State diagram for the message notification mechanism.**

*SIGIO signal handler*
– reset counter
– disable SIGIO signals
– enable timer for regular polling
– switch to communication thread to receive the message that generated the SIGIO signal

Receiving messages immediately on their arrival could lead to a large number of messages in memory if the application does not "consume" the messages. We set a threshold for the maximum number of messages that can be buffered in memory. If this threshold is exceeded, then we disable message receives and the compute thread alone is allowed to proceed (state S4 in Fig 1). The compute thread is expected to consume messages in the course of its computation. When a program enters this stage neither polling nor SIGIO is active, but messages will not be lost since the underlying PVM and the daemons together ensure that messages will not be missed. Once the number of buffered messages falls below the threshold, we resume receiving messages immediately on their arrival by enabling SIGIO (we move from state S4 to state S1 in Fig 1).

By disabling polling and enabling SIGIO after $N_{poll}$ switches (the state transition from *S*2 to *S*1), we avoid wasting time in polling when there are no messages to be received. The SIGIO signal handler restarts the polling mechanism only when a new message arrives at the network. So, at any point in time only one of the two notification mechanisms is active, and by controlling the $N_{poll}$ parameter we can control the "mix" of the two mechanisms. If we set $N_{poll}$ very high then the system becomes almost a polling mechanism, and if we set $N_{poll}$ to zero then we make the system fully interrupt driven. The combination of signaling and polling mechanisms is particularly useful in applications that have a bursty communication pattern [8]. Usually polling is cheaper than signal handling [4]. In systems which allow efficient signal handling one may want to use the interrupt driven mode alone.

Switching between the computation and the communication threads, requires us to save not only the processor context, but also the context of the communication functions in PVM. The context information that needs to be saved in PVM is the current send and receive message buffers.

## 4. EXPERIMENTS

This section discusses the experimental setup, the hardware, software platforms and the programs used to test the enhanced PVM. We use two illustrative applications, matrix multiplication and sorting, to evaluate PVM-AM. In both test applications we compare the "wall-clock" time used to solve a problem of a given size, and the amount of time the application spends in receiving messages. All communication between processors uses the fastest mode available in PVM as explained in the last paragraph of Section 2.

### 4.1 Computing Environment

The laboratory setup consists of a set of 8 HP9000/800 workstations connected by a 10Mbps Ethernet network. Of the 8 machines we used 5 to test our implementation of PVM-AM. Each workstation has its own local disk space, and is connected to a NFS file server. The workstations use the HP-UX 9.04 operating system. Each workstation has a PA-RISC 1.1 CPU and 64MB RAM.

### 4.2 Test Cases

We compare the performance of PVM-AM and regular PVM using two test programs, matrix multiplication and sorting. These two test cases represent computation intensive and communication intensive applications respectively. In case of matrix multiplication the communication time is of the order $O(n^2)$ and the computation time is of the order $O(n^3)$. So the ratio of computation to communication is of the order $O(n)$. In the case of sorting this ratio is of the order $O(log(n))$. Another factor in choosing these two programs is they exhibit very different communication patterns. By "pattern of communication" we mean when and from where messages are sent. In the case of matrix multiplication we can predict the communication pattern before hand, but in the case of parallel sorting, the communication pattern is dependent on the data to be sorted.

#### 4.2.1 Matrix Multiplication

For the operation $C = A * B$, the parallel algorithm partitions the $B$ matrix into four parts $B_i$, each containing $N$ rows and $N/4$ columns. $B_i$ is sent to processor $P_i$. The $A$ matrix is broadcast to all the four processors in blocks of $k$ rows. This is programmed in a typical master slave style with the master sending the data and waiting for results, while the slaves wait for input data, perform the operation and send back a "done" message. Each slave starts up and then receives the $B_i$ matrix. Each slave computes its portion of the result matrix i.e. $C_i = A * B_i$. The $A$ matrix is broadcast by the master in blocks of $k$ rows at a time. Each slave processor $P_i$ waits for a block of $k$ rows of the $A$ matrix from the master, then uses these rows to compute the partial product $C_i$. This step is performed until each $P_i$ has calculated $C_i$. The slaves then send back the $C_i$ matrix and the master assembles the complete $C$ matrix. Figure 2 shows the layout of data and the computations performed at each node. We implemented two versions of the matrix multiplication algorithm, one using regular PVM (MM) and the other using PVM-AM (AMM). The two programs are exactly the same, the only difference being that AMM uses the communication primitives discussed in Appendix, while the AMM version uses the regular PVM communication primitives. The AMM version uses $N_{poll} = 1$ and the time slice devoted to the communication thread during polling is 200ms. We use five workstations, one as master and the other four as slaves. Execution time is measured as the time elapsed from when the master sends the first piece of data, to when it receives a "done" message from the last slave.

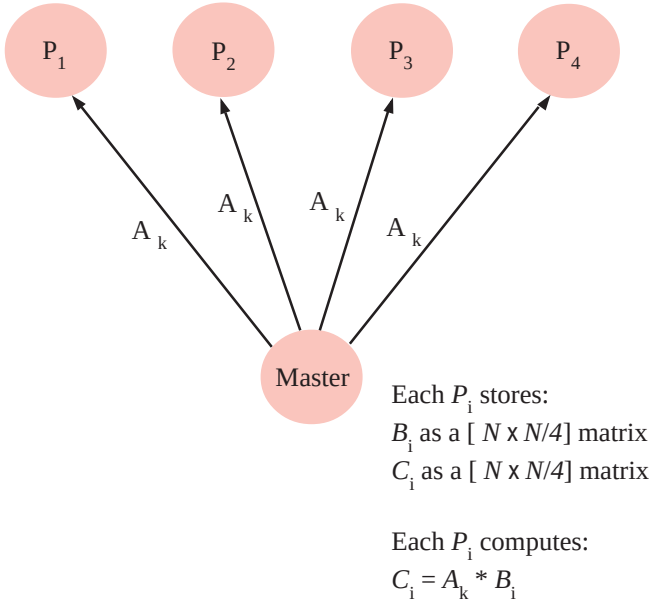



**Figure 2. Data layout for matrix multiplication**

#### 4.2.2 Sorting

The sorting program sorts a file of randomly generated characters in ascending order. The sorting is performed by building a sorting network of processes, the "leaf" processes read the input file and sort their portion of the file using quicksort. The leaf processes then send out the sorted section of the input file to the next higher level of processes. The non-leaf processes merge the data from the left and right children and pass it on to the next higher level. The root process merges the data from its left and right children and writes the sorted data onto a file. The sorting network layout is shown in Figure 3. We have two versions of the sorting program, SORT which uses regular PVM communication primitives and ASORT which uses PVM-AM communication primitives. The ASORT version uses $N_{poll} = 3$ and the time slice for the communication thread during polling is 200ms.

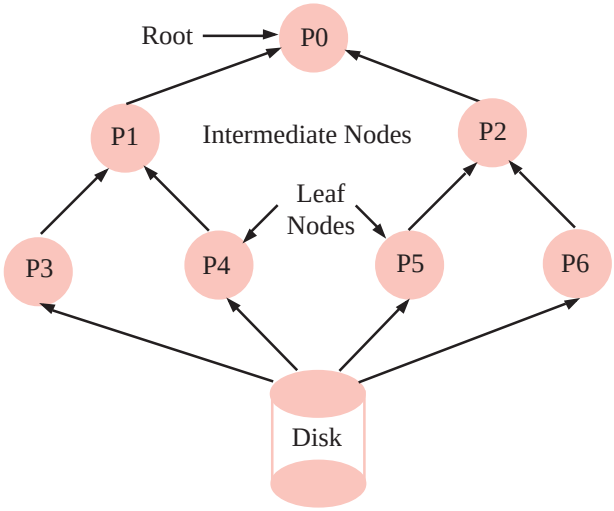



**Figure 3. Logical view of the sorting network**

## 5. RESULTS

The experiments discussed in Section 4 were run in dedicated environment where no other user processes were allowed to run on the machines. The communication time and the total execution time for matrix multiplication are shown in Tables I and II respectively. Tables III and IV give the communication time and the total execution time respectively for the parallel sorting program. The timings were found to be consistent when the experiments were repeated several times. For the multiple runs, the variation in the *communication time* for both sorting and matrix multiplication were around 2%. The variations in the *execution time* for matrix multiplication and sorting were less than 5% respectively.

In both test cases, the time spent in receiving messages is significantly lower for the version using Active PVM communication primitives compared to the version using the regular PVM communication primitives. In the matrix multiplication program for a problem size of 1408 we get a improvement of 2.1 compared to 1.4 for other sizes. It is unclear why we get a distinctly higher improvement factor for this case alone.

Since sorting is communication intensive, the reduction in communication time leads to a significant improvement in the total execution time for ASORT (which uses Active PVM) compared to the total execution time for SORT (which uses regular PVM).

## 6. CONCLUSION

In this paper we have presented a new communication scheme to improve the performance in PVM. We have shown that the new communication scheme significantly reduces the communication time. In comparison to the PVM library PVM-AM results in 20% to 50% improvements in the communication performance in test cases of matrix multiplication and sorting. Existing code written for PVM can be easily ported to use the new PVM-AM library.

## APPENDIX: PROGRAMMER INTERFACE

PVM-AM provides the programmer with the following set of functions:

*pvm_active_init(work fn):*

This function should be called after tasks have been

**TABLE I**
**Comparison of Message Receive Times for Matrix Multiplication**

| Matrix Size | No. of Mesgs. | Matrix Multiply using Active PVM | Matrix Multiply using normal PVM | Improvement Factor Due to Active Messages |
|---|---|---|---|---|
| | | Mesg. Recv. Time | Mesg. Recv. Time | |
| 1024 | 8 | 13.71 | 18.98 | 1.38 |
| 1280 | 10 | 19.80 | 28.58 | 1.44 |
| 1408 | 11 | 22.98 | 48.07 | 2.09 |
| 1536 | 12 | 31.26 | 42.40 | 1.36 |
| 1664 | 13 | 34.12 | 46.57 | 1.36 |
| 2048 | 16 | 47.76 | 92.34 | 1.93 |

**TABLE II**
**Comparison of Execution Times for Matrix Multiplication**

| Matrix Size | No. of Mesgs. | Matrix Multiply using Active PVM: Execution Time | Matrix Multiply using normal PVM: Execution Time | Improvement Factor Due to Active Messages |
|---|---|---|---|---|
| 1024 | 8 | 363.19 | 362.29 | 1.00 |
| 1280 | 10 | 710.10 | 715.35 | 1.01 |
| 1408 | 11 | 947.95 | 973.35 | 1.03 |
| 1536 | 12 | 1229.40 | 1242.84 | 1.01 |
| 1664 | 13 | 1582.04 | 1601.14 | 1.01 |
| 2048 | 16 | 2980.47 | 3056.31 | 1.03 |

spawned, to initialize the AM code and to start the threads. The work_fn is the function that does all the computation in the application. The threading mechanism will switch between this function and the internal communication thread using a timer to generate SIGALRM as explained in Section 3.

*pvm_active_send(tid, tag)*:

This function has the same functionality as the usual PVM pvm send(tid,tag) function, i.e. it sends the message to task *tid* with a tag *tag*.

*pvm_active_receive(tid, tag)*:

This function has the same functionality as the usual PVM pvm_recv(tid,tag) function, i.e. it receives a message with tag *tag* from task *tid*. This function searches the set of previously received messages to see if the requested message is in memory. If the requested message is not found we switch to the communication thread and wait for a message to arrive.

*pvm_active_set_do_only_work(tag, N)*:

This function enables the user to specify that if more than *N* messages with tag *tag* are pending in memory, then the application wants to do only computation. This function enables the user to "consume" messages quickly and prevents too many messages from building up and using memory. This parameter needs to be set by the programmer based on the message size used in communication.

*pvm_active_set npoll(N)*:

This function sets the $N_{poll}$ value for the communication thread. The $N_{poll}$ value controls the number of polls performed by the communication thread as explained in Section 3.

*pvm_active_exit( )*:

This is a cleanup function to exit from PVM-AM and performs house-keeping activities to ensure a proper exit from PVM-AM.

Existing PVM applications can be adapted to use the AM communication library as follows:

**TABLE III**
**Comparison of Message Receive Times for Parallel Sorting**

| Input File Size | No. of Mesgs. | Parallel Sorting using Active PVM: Mesg. Recv. Time | Parallel Sorting using regular PVM: Mesg. Recv. Time | Improvement Factor Due to Active Messages |
|---|---|---|---|---|
| 2097152 | 512 | 10.89 | 13.29 | 1.22 |
| 4194304 | 1024 | 24.44 | 31.63 | 1.29 |
| 8388608 | 2048 | 48.26 | 67.26 | 1.39 |
| 16777216 | 4096 | 91.37 | 140.52 | 1.54 |

**TABLE IV**
**Comparison of Execution Times for Parallel Sorting**

| Input File Size | No. of Mesgs. | Parallel Sorting using Active PVM: Mesg. Recv. Time | Parallel Sorting using regular PVM: Mesg. Recv. Time | Improvement Factor Due to Active Messages |
|---|---|---|---|---|
| 2097152 | 512 | 16.60 | 16.74 | 1.01 |
| 4194304 | 1024 | 35.60 | 38.37 | 1.08 |
| 8388608 | 2048 | 66.57 | 80.74 | 1.21 |
| 16777216 | 4096 | 138.81 | 167.62 | 1.21 |

- Call AM_init(work) immediately after spawning child tasks
- The function *work*( ) performs all the computation in the application
- Change pvm_send(tid, tag) function calls to pvm_active _send(tid, tag)
- Change pvm_recv(tid, tag) function calls to pvm_active _recv(tid, tag)
- Change pvm_exit( ) function calls to pvm_active_exit ( )

The main point to be kept in mind while porting code to PVM-AM is that the "main" function starts the child tasks and then calls the "pvm_active_init" function.

## References


[1] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, Steve Otto, and Jon Walpole. PVM: Experiences, current status and future direction. *Supercomputing '93 Proceedings*, 1993.

[2] G.A.Geist and V.S.Sunderam. Network based concurrent computing on the PVM system. *Concurrency: Practice and Experience*, 4(4), June 1992.

[3] K.E.Schauser and C.J.Scheiman. Active Messages implemen tations for the Meiko CS-2. Technical report, Department of Computer Science, UC Santa Barbara, October 1994.

[4] Lok T. Liu and David E. Culler. Measurements of Active Messages performance on the CM-5. Technical report, Department of Computer Science, UC Berkeley, May 1994.

[5] L.W.Tucker and M.Mainwaring. CMMD:Active messages on the CM-5. *Parallel Computing*, 20(4), April 1994.

[6] Richard P. Martin. HPAM: An active message layer for a network of HP workstations. *Proceedings of Hot Interconnects II*, August 1994.

[7] Philip J. Mucci and Jack Dongarra. Possibilities for Active Messaging in PVM. Technical report, University of Tennessee, February 1995.

[8] W. Richard Stevens. *Unix Network Programming*. Prentice Hall, 1990.

[9] S.White, A.Alund, and V.S.Sunderam. Performance of the NAS parallel benchmarks on PVM-based networks. *Journal of Parallel and Distributed Computing*, 26(1), April 1995.

[10] Thorsten von Eicken. Building parallel programming languages using active messages. Technical report, Depart ment of Computer Science, Cornell University, 1994.

[11] Thorsten von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. ActiveMessages: A mechanism for integrated communication and computation. *Proceedings of the 19th International Symposium of Computer Architecture*, May 1992.



**KRISHANAN R. SUBRAMANIAM** got his B.E. from the Coimbatore Institute of Technology in India. He got his M.S. in computer science from Iowa State University. He is currently working for Apple Computers. His research interests are parallel and distributed computing and operating systems.

**SURAJ C. KOTHARI** got his B.Sc. from Poona University in India and Ph.D. in mathematics from Purdue University. He is currently a professor of computer science at Iowa State University. His research interests are parallel and distributed computing, scientific computing, and neural networks. His group is developing two experimental systems: PARALLELIZATION AGENT, a system for class-specific automatic parallelization, and BATRUN, a distributed batch processing system to utilize idle computers in a network as a resource for large-scale computing.

**DON E. HELLER** got his B.S. in mathematics and Ph.D. in computer science, both from Carnegie-Mellon University. He is currently a senior scientist at Ames Laboratory (U.S. DOE), and an adjunct associate professor of computer science at Iowa State University. Previously, he has held positions at the Institute for Computer Applications in Science and Engineering, Lawrence Livermore National Laboratory, Pennsylvania State University, Bellaire Research Center (Shell Oil), and the Center for Research on Parallel Computation at Rice University. His research interests are parallel and distributed computing and scientific computing.